Cahier des charges

-

Éditeur de Fichiers Coopératif et Réparti

Corentin LEVY 3770522

Bastien MASSON 3502283

1er mars 2018

Encadré par Pierre Sens

Le but de ce projet est de développer un éditeur de fichiers coopératif et réparti permettant à un ensemble d'utilisateurs de travailler simultanément sur un même document. Ce projet est donc découpé en deux parties : un serveur de fichiers ainsi qu'un mini-éditeur de fichiers. Le projet sera réalisé intégralement en langage C.

Table des matières

1 Besoins

Cette section décrira précisément ce qui est attendu dans ce projet.

Comme dit précédemment, il s'agit de réaliser un éditeur de fichiers, qui possède la particularité d'être coopératif (et donc implicitement réparti), c'est à dire que plusieurs utilisateurs peuvent travailler en même temps sur le même document. Un exemple concret d'un tel éditeur est **Google Docs**.

Le projet est donc découpé en deux parties majeures : un serveur de fichiers, ainsi qu'un mini-éditeur de texte.

Le serveur de fichiers est dit "centralisé" (serveur maître). Lancé sur une seule machine, son rôle est d'enregistrer les fichiers des éditeurs sur le disque, de répondre à leurs requêtes selon un protocole défini à l'avance, et enfin, implémenter une politique de cohérence pour les fichiers. La politique de cohérence sera la suivante :

— UPDATE : Après chaque modification d'une ligne, la ligne sera renvoyée par le serveur à tous les clients qui ont ouvert ce fichier.

De plus, chaque éditeur maintiendra localement une copie locale des fichiers en cours d'utilisation par l'utilisateur. Pour assurer la cohérence des données, il faudra implémenter un mécanisme de verrouillage de ligne : lorsqu'une ligne est en cours d'édition, les autres clients ne peuvent pas la mofifier.

L'interface du mini-éditeur de texte permettra de :
— Créer un fichier ;
— Supprimer un fichier ;
— Lire un fichier ;
— Modifier (ligne par ligne) un fichier.

Cet éditeur sera contrôlable avec des commandes, ainsi qu'avec le clavier, un peu à la manière de Vim. Cela est fait pour éviter d'avoir recours à des bibliothèques graphiques trop lourdes qui pourraient impacter le reste du projet. Par conséquent on aura à définir non pas un mais deux protocoles. Le premier, totalement invisible pour l'utilisateur, servira à la communication

entre le serveur et le client. Le second, visible pour l'utilisateur, servira à contrôler l'éditeur de texte, à la manière de Vim par exemple.

2 Conditions d'utilisation

Notre programme se voudra simple d'utilisation, intuitif et efficace pour l'utilisateur. Il sera utilisable uniquement avec des commandes, c'est à dire qu'il n'y aura pas d'interface utilisable avec la souris par exemple.

Un client peut se connecter, éditer des fichiers, demander la liste des fichiers ainsi que la liste de qui est connecté au serveur.
Le serveur quant à lui se contente de répondre aux requêtes des clients, ainsi que d'envoyer à intervalles réguliers des messages de type "ping" à tous les clients sur un deuxième port en broadcast UDP pour vérifier qu'un client est toujours connecté. Cela permet d'éviter qu'un client en panne bloque un fichier pour toujours.
Étant donné qu'un message envoyé en UDP n'a aucune certitude d'être reçu, on conviendra d'un nombre arbitraire de messages non reçus au delà duquel un client est considéré comme en panne.

Sera décrit ici un cas d'utilisation typique du projet. Toutefois, on ne rentrera pas ici dans les aspects techniques, il s'agit uniquement de décrire un scénario d'utilisation type de notre future application.

Scénario type où le client veut créer un fichier, le remplir, l'enregistrer puis quitter (tous les messages échangés entre le serveur et le client suivent le protocole du programme invisible pour le client qui sera défini plus loin, à l'exception de ceux pour éditer le fichier) :

1. Le client lance le programme
2. Le programme se connecte au serveur et indique au client qu'il peut commencer à travailler
3. Le client envoie un message au serveur pour créer un fichier
4. Le serveur confirme la bonne création du fichier
5. Le client envoie un message au serveur pour modifier le fichier créé
6. Le serveur ouvre le fichier et renvoie son contenu au client en ouvrant l'éditeur de texte

7. A l'aide de commandes, le client édite le fichier, puis demande au serveur de le sauvegarder
8. Le serveur confirme la sauvegarde du fichier
9. Le client quitte le programme
10. Le serveur nettoie les traces du passage du client en conséquence

Scénarios alternatifs :

— Si un client est déjà connecté et en train d'éditer le même fichier, alors la ligne sur laquelle il est en train de travailler apparaît comme verrouillée pour les autre clients (à l'aide d'un symbole quelconque), et donc non modifiable tant que ledit client n'a pas soit quitté, soit envoyé ses modifications au serveur.

— Si un client veut modifier un fichier qui n'existe pas, le serveur le créé automatiquement en prévenant le client auparavant, à la manière de Vim, Gedit, etc.

— Un client ne peut pas bloquer indéfiniment une ligne dans l'hypothèse d'une déconnexion pendant la modification d'une ligne. Si un tel évènement se produit, le client déconnecté est automatiquement relevé de ce qu'il était en train de faire et la ligne est dévérouillée. Les modifications du client déconnecté sont sauvegardées dans un fichier spécial, ainsi il pourra récupérer son travail lors de sa reconnexion.

3 Architecture Logicielle

3.1 Choix techniques

Comme dit au début de ce document, l'intégralité de ce projet sera réalisé en C. Il y aura deux types d'entités distinctes, les clients et le serveur. Le serveur étant un "maître", il ne pourra exister qu'une seule instance à la fois.

Général :

Le protocole de communication possède les propriétés suivantes : chaque entête de requête qui est une demande sera suivie d'un point d'interrogation " ?", tandis que chaque entête de requête qui sera une réponse sera suivie d'un point d'exclamation " !".

Les noms de fichiers enregistrés sur le serveur ne peuvent contenir de caractère espace car cela irait à l'encontre du protocole de communication défini plus bas.

La partie graphique sera réalisée à l'aide de la bibliothèque Ncurses, qui permet de réaliser des interfaces simples en console. Les interactions entre l'interface et l'utilisateur seront exclusivement en ligne de commande et non avec la souris. Le clavier sera utilisé pour naviguer dans le document, en envoyant une requête correspondante à la touche appuyée par l'utilisateur (ex : flèche bas → on navigue vers le bas).

Client :
Un client est composé de deux threads et de deux ports : un port TCP associé au premier thread qui servira aux communications directes avec le serveur, ainsi qu'un port UDP associé au deuxième thread qui servira à recevoir les "ping" du serveur. Chaque client garde en local une copie des fichiers en cours de modification, qui seront supprimés lors de la déconnexion. Ainsi, si lors d'une connexion, le programme détecte la présence de fichiers temporaires on pourra assumer que le client s'est mal déconnecté, et ainsi proposer d'enregistrer ou non le contenu de ces fichiers.

Voici la liste des différentes commandes disponibles pour le client, **hors édition de fichier**. On entend par là les commandes qui permettent de créer, modifier, supprimer un ficher, quitter le programme, afficher la liste des fichiers ou des utilisateurs. Ce dernier peut utiliser ces commandes pour utiliser l'application à sa guise. Elles seront ensuite transformées vers le format du protocole client - serveur :

- **create nom_de_fichier** → créer un fichier
- **modify nom_de_fichier** → modifier un fichier
- **delete nom_de_fichier** → supprimer un fichier
- **quit** OU **exit** → quitter le programme
- **listf** → afficher la liste des fichiers
- **listu** → afficher la liste des clients
- **help** → afficher un texte d'aide (commande locale, n'est pas envoyée vers le serveur)

Voici les caractéristiques du protocole utilisé pour la communication dans

le sens client - serveur, invisible à l'utilisateur :

Chaque message sera composé de la manière suivante :

— Un entête de 3 lettres minuscules représentant le type du message suivi d'un point d'interrogation " ?", à l'exception du message de réponse au "ping" du serveur, **png !**, qui sera suivi d'un point d'exclamation. Ce type peut etre : **con ?, qui ?, cre ?, del ?, mod ?, lfi ?, lst ?, png !, scd ?, scu ?**, pour respectivement se connecter, quitter, créer, supprimer, modifier un fichier, sauvegarder une ligne, afficher la liste des fichiers du serveur, afficher les clients connectés au serveur, répondre à un "ping" du serveur, naviguer le document vers le bas (scroll down), naviguer le document vers le haut (scroll up).
Notons qu'il n'y a pas d'entête pour demander à sauvegarder un fichier. C'est en effet inutile, la sauvegarde sera faite à chaque envoi de ligne au serveur.

— Un simple espace, excepté pour **con ?, qui ?, lfi ?, lst ?, png !** qui seront envoyés tels quels.

— Pour les autres, un nom de fichier.

Chacun de ses messages sera envoyé à l'initiative de l'utilisateur, à l'exception du **png !** qui sera une réponse automatique à un message du serveur vérifiant que le client est toujours connecté correctement.

Voici à présent les caractéristiques du protocole utilisé lors de l'édition d'un fichier.

Les différents entêtes peuvent être : **mlg ?, dlg ?, ilg ?, sav ?, scd ?, scu ?**

Nous mettrons ici en parallèle la commande tapée par l'utilisateur ainsi que sa transcription en protocole client - serveur. On suppose qu'au moment de taper ces commandes, l'utilisateur visualise le document :

Pour modifier une ligne :
m 15 → "Je voudrais modifier la ligne 15" → **mlg ? 15 texte de la ligne**

Pour supprimer une ligne :
d 10 → "Je voudrais supprimer la ligne 10" → **dlg ? 10**

Pour ajouter une ligne à la fin du document :
i → "Je voudrais ajouter une ligne à la fin du document" → **ilg ?**

Pour ajouter une ligne à un endroit précis :
i 5 → "Je voudrais ajouter une ligne après la ligne 4" → **ilg ? 5**

Pour sauvegarder une ligne modifiée :
appui sur la touche "Entrée" → "Je voudrais valider ma modification de ligne" → **sav ? la ligne modifiée**

Pour naviguer le document vers le bas :
appui sur la touche "flèche bas" → "Je voudrais naviguer vers le bas" → **scd ?**

Pour naviguer le document vers le haut :
appui sur la touche "flèche haut" → "Je voudrais naviguer vers le haut" → **scu ?**

Voici un résumé des entêtes possibles pour le protocole client - serveur : **con ?, qui ?, cre ?, del ?, mod ?, sav ?, lfi ?, lst ?, png !, mlg ?, dlg ?, ilg ?, scd ?, scu ?**

Serveur :
Un serveur est composé de deux ports : un port TCP pour accepter les connexions, et un port UDP en broadcast utilisé pour envoyer des "ping" aux clients. Le serveur est composé d'un thread principal qui s'occupe d'accepter les connexions des clients, un thread secondaire qui s'occupe d'envoyer les messages de "ping" à tous les clients connectés, ainsi que d'un thread supplémentaire par client qui gère les requêtes spécifiques à ce client.

Á chaque instant le serveur doit savoir quel client modifie quel fichier. Il se sert de cette information pour répondre correctement à toutes les requêtes du client qui travaille sur ledit fichier.

Voici les caractéristiques du protocole utilisé pour la communication dans le sens serveur - client, invisible à l'utilisateur :

Chaque message sera composé de la manière suivante :

— Un entête de 3 lettres minuscules représentant le type du message suivi d'un point d'exclamation " !", à l'exception du message de demande aux clients "ping", **png !**, qui sera suivi d'un point d'interrogation. Ce type peut etre : **con !, qui !, cre !, del !, mod !, lfi !, lst !, png ?, scd !, scu !, err !**, pour respectivement confirmer la connexion, confirmer qu'un client quitte proprement, confirmer la création d'un fichier, confirmer la suppression d'un fichier, confirmer l'ouverture d'un fichier à modifier, renvoyer la liste des fichiers du serveur, renvoyer la liste des clients connectés au serveur, envoyer un "ping" à tous les clients, envoyer la bonne ligne pour que le client navigue le document vers le bas (scroll down), envoyer la bonne ligne pour que le client navigue le document vers le haut (scroll up), et enfin si une erreur est survenue.
— Un simple espace, excepté pour **con !, qui !, png ?** qui seront envoyés tels quels, et pour **mod !, lfi !, lst !** qui nécessitent un entier collé à l'entête.
— **mod !** est suivi d'un nombre X indiquant le nombre de lignes restant à envoyer. Ce nombre arbitraire X sera défini à l'avance et sera le nombre de lignes maximum à afficher à la fois sur l'écran. Une réponse **mod !X** sera donc suivie par (X - 1) autres réponses contenant chacunes une ligne à afficher. De cette façon on obtient un affichage du fichier demandé pour le client. On envoie ensuite la réponse.
— **cre !, del !, mod !** sont tous les trois suivis d'un nom de fichier et envoyés tels quels.
— **lfi !, lst !** sont suivis d'un nombre X indiquant le nombre d'éléments restants à envoyer. Une réponse **lfi !X OU lst !X** sera donc suivie par (X - 1) autres réponses contenant chacunes un élément (soit un fichier soit un utilisateur). On envoie ensuite la réponse.
— **err !** est suivi d'une chaîne de caractères expliquant l'erreur qui est survenue. On envoie ensuite la réponse.

3.2 Découpage du code

Le code sera découpé assez logiquement de la façon suivante, le but étant de rendre une architecture simple et compréhensible qui facilitera la lecture du code, la programmation et le débug :

Les fichiers seront déposés dans différents dossiers : un pour le client, un pour le serveur, et un qui rassemblera les fichiers nécessaires aux deux entités.

Coté client, un fichier principal contiendra la fonction main et qui se chargera d'appeler toutes les autres fonctions, un deuxième fichier contiendra les méthodes nécessaires au formatage et au déformatage des messages provenant et à destination du serveur. Un troisième fichier s'occupera du port UDP et donc de la réception des "ping" du serveur. Bien entendu, un header contenant les signatures, les imports, les macros et les définitions de structures sera présent.
Ces fichier pourront potentiellement s'appeler, respectivement :

- client_main.c
- formatage.c
- client_udp.c
- client_header.h

Ils seront contenus dans le dossier client à l'exception de formatage.c qui ira dans le dosser commun aux deux entités.

Côté serveur, un fichier principal contiendra également une fonction main. Il sera possible de réutiliser le même fichier contenant les fonctions pour formater et déformater les messages du protocole défini plus haut. Un fichier sera consacré aux connexions clients, il appellera un autre fichier qui s'occupera du client qui vient de se connecter. Un header sera aussi inclu. Cela nous donne donc :

- serveur_main.c
- formatage.c
- serveur_accept.c
- serveur_traite_client.c
- serveur_header.h

Ces fichiers seront contenus dans le dossier serveur à l'exception de formatage.c.

3.3 Structures

Dans cette section seront décrit des idées de structures pour implémenter notre programme. Elles sont sujettes à améliorations et ne sont en aucun cas finales, car pouvant évoluer avec les besoins du programme.

Nous pourrons utiliser une structure pour stocker les informations d'un client qui se connecte. Ainsi, on garde un moyen simple de l'identifier et

d'obtenir certaines informations utiles.

```
typedef struct {
        int port;                //port of user
        int ip;                        //ip of user
        int nb_open_files;        //number of files the user has open this session
        int is_modifying;        //if the user is modifying something
        int line_nb;                //number of the line the user is modifying
        int unanswered_pings;   //if this reach a maximum, user is considered AFK
} client;
```

Une structure pour garder des informations utiles sur les fichiers du serveur (pour éviter que l'on ait besoin de lire à chaque fois le contenu du dossier courant par exemple) :

```
typedef struct {
        int nb_of_files;
        int nb_of_open_files;
        open_lines current_lines[]; //struct containing the open files
                                    //and the line being edited
        client clients_editing[]; //list of editing clients
} files;
```

Et la structure open_lines associée :

```
typedef struct {
        char *name;
        int line_nb;
} open_lines;
```

4 Conditions de validation

On prendra soin de déclarer un fichier de tests s'assurant du bon fonctionnement du programme. Il comprendra des tests sur l'ensemble des méthodes écrites et permettra de vérifier aisément que le travail a correctement été réalisé. Rédigé au fur et à mesure, il sera ensuite appliqué à chaque amélioration afin d'assurer la rétro-compatibilité des nouveautés implémentées.

4.1 Résultats attendus

On s'attend à l'issue de la phase de code de ce projet à obtenir un programme fonctionnel, avec le moins de bugs possible (aucun programme n'est parfait, malheureusement). On s'attend également à ce que tous les points évoqués dans la partie **Conditions d'utilisation** soit remplis. Une tolérance minime aux pannes sera mise en place, potentiellement améliorée si le temps le permet.

Un certains nombres d'outils permettant l'administration du programme seront également mis en place, c'est à dire des logs, des stats et du monitoring. Le but est bien entendu de rendre le travail du programmeur ainsi que celui du correcteur plus facile.

4.2 Améliorations potentielles

Les améliorations dans cette section seront implémentées dans la mesure du possible et suivant l'avancement global du projet. Elles contribueront à avoir un programme plus résistant, plus souple et plus agréable à utiliser. L'ordre n'est pas significatif.

→ Quand on modifie une ligne et qu'on appuie sur "Entrée", modifie automatiquement la ligne suivante sans avoir à renvoyer une requête.

→ Quand on a planté et qu'on se reconnecte, le client détecte les fichiers temporaires non effacés, demande à l'utilisateur si il désire récuperer leur contenu, puis envoie au serveur les bonnes instructions pour effectuer ces changements (amélioration ambitieuse).

→ Ajout d'une commande pour se déplacer directement à une ligne précise, pour éviter d'avoir à envoyer plusieurs commandes pour se déplacer à la suite (à la manière de Vim).

→ Ajout d'une commande pour accéder aux logs directement depuis le programme.

→ Ajout d'une commande permettant de faire une sauvegarde des fichiers du serveur chez le client.

4.3 État de l'art

Pour ce projet nous nous sommes inspirés de quelques éditeurs de texte très connus. Google Docs pour la partie répartie, Vim pour la partie commandes. Pour le protocole nous nous sommes basés sur des projets réalisés par le passé, notamment en L3 (à Paris 7). La bibliothèque graphique Ncurses nous a été suggérée par Pierre Sens.